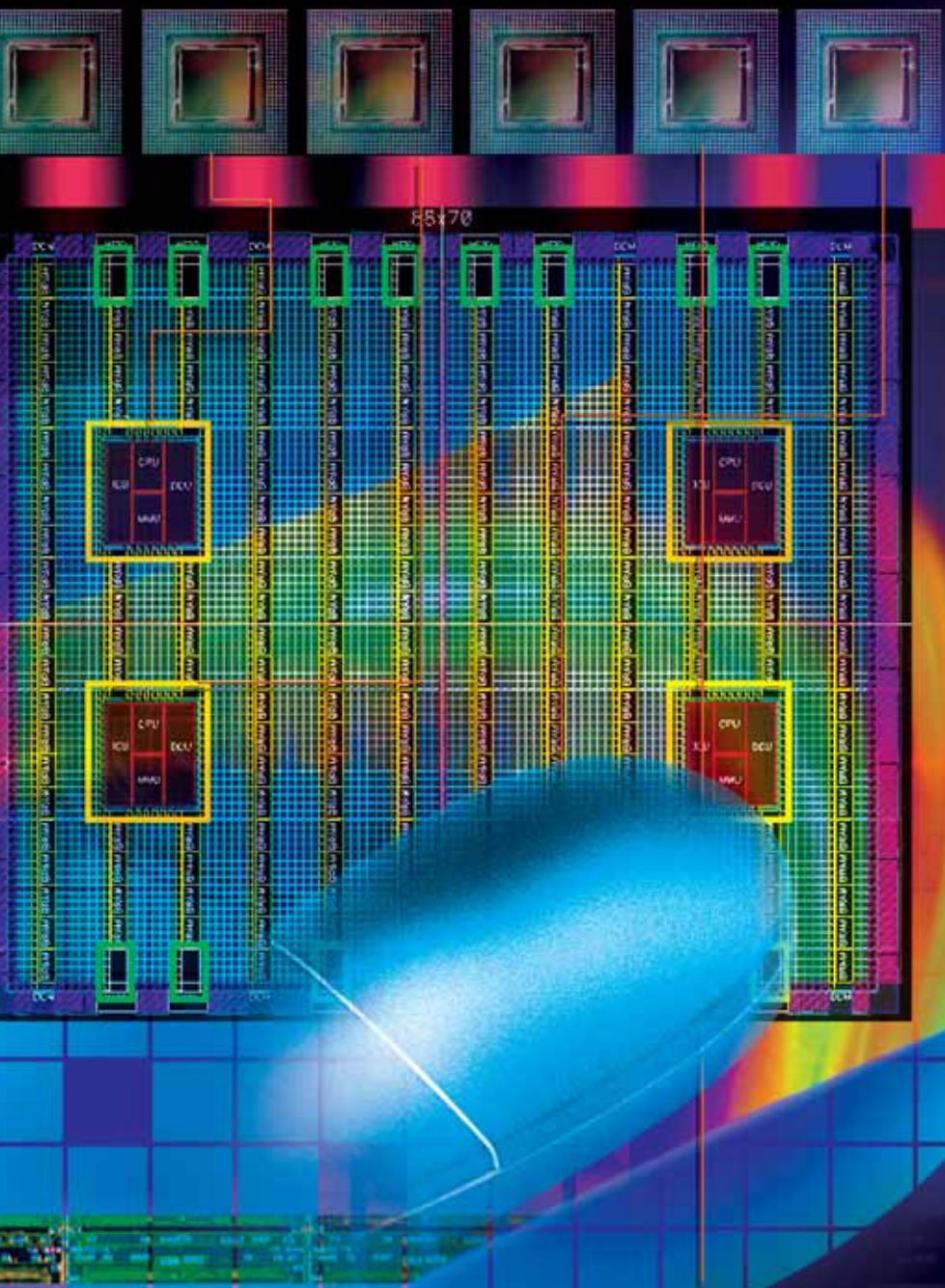


Accelerating System Performance Using ESL Design Tools and FPGAs

ESL design tools provide the keys for opening more applications to algorithm acceleration.



by James Hrica
Systems Architect
Celoxica Inc.
james.hrica@celoxica.com

Jeff Jussel
GM Americas
Celoxica Inc.
jeff.jussel@celoxica.com

Chris Sullivan
Strategic Marketing Director
Celoxica Inc.
chris.sullivan@celoxica.com

The microprocessor has had more impact on electronics and our society than any other invention since the integrated circuit. The CPU's programmability using simple sets of instructions makes the capabilities of silicon available to a broad range of applications. Thanks to the scalability of silicon technologies, microprocessors have continuously grown in performance for the last 30 years.

But what if your processor does not have enough performance to power your application? Adding more CPUs may not fit power or cost budgets – and more than likely won't provide the needed acceleration anyway. Adding custom hardware accelerators as co-processors adds performance at lower power. But unlike processors, custom hardware is not easily programmable. Hardware design requires special expertise, months of design time, and costly NRE development charges.

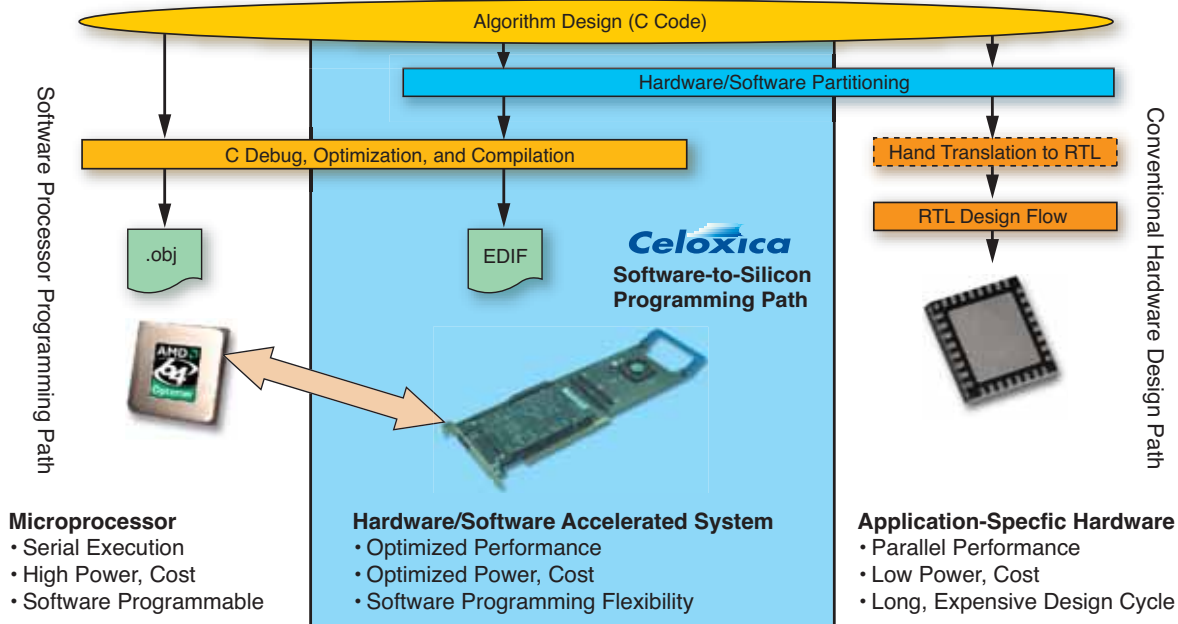


Figure 1 – Software design flows directly program FPGAs.

Enter the combination of high-performance Xilinx® FPGAs and electronic system-level (ESL) design tools, expressly tuned for FPGA design. By taking advantage of the parallelism of hardware, certain algorithms can gain faster performance – at lower clock speeds using less power – than what is possible in serial processor operations. Xilinx FPGAs provide a reconfigurable option for hardware acceleration, while the addition of ESL design flows from Celoxica gives developers access to these devices using a software design flow. Software-based design flows are now able to directly program Xilinx FPGA devices as custom co-processors (see Figure 1).

from parallelism. Specifically applied to systems using multiple CPUs, Amdahl's law states that the acceleration possible is limited to the proportion of the accelerating algorithm's total processing time. If the algorithm takes up p% of the total processing time and is spread across N processors, then the potential speedup can be written as $1/[(1-p) + p/N]$. For example, assuming that 20% of an algorithm is spread across four processors, the total processing time is at best reduced to 85% of the original. Given the additional cost and power required by three extra processors, a mere 1.2x speedup may not provide a big enough return.

Using custom hardware accelerators, the N in Amdahl's equation becomes the speedup multiplier for accelerated algorithms. In hardware, this multiplier can be orders of magnitude higher; the power utilization and costs of FPGA accelerators are generally much lower. Still, Amdahl's law would still seem to limit the potential theoretical total acceleration times. Fortunately, in practice Amdahl's law has proven too pessimistic; some suggest that the proportional throughput of parallel hardware increases with additional processing power, making the serial portion smaller in comparison. For certain systems, total accelerations of one, two, or even three orders of magnitude are possible.

Will FPGA Acceleration Work for All Algorithms?

When a general-purpose CPU or DSP alone can't deliver the performance, several options are available (Figure 2). However, not all algorithms lend themselves well to all types of acceleration. The first step in the process is to determine whether the application will benefit from the parallelism of hardware.

Amdahl's law, developed by computer architect Gene Amdahl in 1967, asserts a pessimistic view of the acceleration possible

Processor/DSP	Xilinx FPGA	Full Custom Hardware
Sequential Execution	Parallel Execution	Parallel Execution
Very Flexible	Flexible	Inflexible
Power-Inefficient	Power-Efficient	Very Power-Efficient
Inexpensive	Relatively Inexpensive (Compared to Custom ASIC)	Very Expensive (Very High Volumes Only)
* C-Based Design for Software and Hardware		
Configurable μ P/FPGA/Programmable SOC/DSP		

Figure 2 – System acceleration possibilities

What Portions of the Algorithm Should Go in the FPGA?

To accelerate a software application using FPGAs, you must partition the design between those tasks that will remain on the CPU and those that will run in hardware. The first step in this process is to profile the performance of the algorithm running entirely on the CPU.

You can gain some insight into where the application spends most of its run time using standard software profiling tools. The profiler report points to the functions, or even lines of code, that are performance bottlenecks. Free profiling tools such as gprof for the GNU development environment or Compuware's DevPartner Profiler for Visual Studio are readily available.

You should analyze the application code profile around areas of congestion to determine opportunities for acceleration. You should also look for sub-processes that can benefit from the parallelism of hardware. To accelerate the system, you move portions of your design into hardware connected to the CPU, such as an FPGA situated on a PCI, HyperTransport, or PCI Express interface board. PowerPC™ or MicroBlaze™ embedded processors provide another possible approach, putting the reconfigurable fabric and processor within the same device.

It is important during the analysis process not to overlook the issues of data transfer and memory bandwidth. Without careful consideration, data transfer overhead can eat into gains made in processing time. Often you may need to expand the scope of the custom hardware implementation to minimize the amount of data to be transferred to or from the FPGA. Processes that expand data going into the accelerated block (such as padding or windowing) or processes that contract data coming out of that block (such as down-sampling) should all be included in the FPGA design to reduce communication overhead.

Where Is Parallelism in the Algorithm?

Hardware accelerates systems by turning large serial software functions and performing them in parallel over many smaller processes. This results in a highly optimized, very specific set of hardware func-

tions. ESL tools are an important part of this process, allowing you to insert parallelism from software descriptions, using FPGAs like custom co-processors.

You can exploit parallelism at many levels to move functionality from software into custom hardware. Fine-grained parallelism is achieved by executing many independent, simple statements simultaneously (variable assignments, incrementing counters, basic arithmetic, and others). The pipelining of sequentially dependent statements is another form of fine-grain parallelism, accomplished by arranging individual stages in parallel such that each stage feeds the input of the next stage downstream.

For example, by pipelining the inside of a code loop, including the loop control logic, it may be possible to reduce iterations to only one clock cycle. After priming the pipeline with data, the loop could be made to execute in the number of clock cycles equal to the number of loop iterations (plus the relatively modest latency). This type of optimization often provides the most benefits when accelerating an algorithm in hardware.

Coarse-grained parallelism can also provide acceleration. Larger independent sub-processes like functions or large looped blocks, which run sequentially on the processor, can run simultaneously in custom hardware. Similarly, unrolling loops either fully or partially can add parallelism. In unrolling, the processing logic inside loops is replicated multiple times so that each instance works in parallel on different data to reduce the number of loop iterations. Pipelining large sequentially dependent processes can further remove cycles at the cost of some latency.

Another important form of coarse-grained parallelism occurs when the software and hardware operate in tandem. For example, a system accelerated by transferring frames of data to custom hardware for co-processing might be structured so that while the hardware processes one frame, the software is accomplishing any requisite post-processing of the previous frame, or pre-processing the next frame. This type of coordination and efficiency is made possible by the common development environ-

ments for both the hardware and software provided by ESL design.

Parallelism can also optimize the memory bandwidth in a custom hardware accelerator. For example, the addition operator for two vectors may be written as a loop, which reads an element of each vector from memory, adds the operands, and writes the resulting sum vector to memory. Iterations of this operation require three accesses to memory, with each access taking at least one cycle to accomplish.

Fortunately, modern Xilinx FPGAs incorporate embedded memory features that can help. By storing each of the three vectors in a separate embedded memory structure, you can fully pipeline the action of the summing loop so that iterations take only one cycle. This simple optimization is one example of how you can maximize system acceleration in hardware.

How Do Accelerators Handle Floating Point?

The use of floating-point mathematics is often the most important issue to resolve when creating custom hardware to accelerate a software application. Many software applications make liberal use of the high-performance floating-point capabilities of modern CPUs, whether the core algorithms require it or not. Although floating-point arithmetic operations can be implemented in PLD hardware, they tend to require a lot of resources. Generally, when facing floating-point acceleration, it is best to either leave those operations in the CPU portion of the design or change those operations to fixed point. Fortunately, you can implement many algorithms effectively using fixed-point mathematics, and there are pre-built floating-point modules for those algorithms that must implement floating point in hardware.

A detailed description of the process for converting floating-point to fixed-point operations can be highly algorithm-dependent, but in summary, the process begins by analyzing the dynamic range of the data going into the algorithm and determining the minimum bit width possible to express that range in an integer form. Given the width of the input data, you can trace through the operations involved to determine the bit growth of the data.

For example, to sum the squares of two 8-bit numbers, the minimum width required to express the result without loss of information is 17 bits (the square of each input requires 16 bits and the sum contributes 1 more bit). By knowing the desired precision of the output, simply work backwards through the operation of the algorithm to deduce the internal bit widths.

You can implement well-designed fixed-point algorithms in FPGAs quite efficiently because you can tailor the width of internal data paths. Once you know the width of the inputs, internal data paths, and outputs, the conversion of data to fixed point is straightforward, leading to efficient implementation on both the hardware and software sides.

Occasionally, an application requires more complex mathematical functions – $\sin()$, $\cos()$, \sqrt{x} , and others – on the hardware side of the partition. For example, these functions may operate on a discrete set of operands or may be invoked inside of loops, with an argument dependent on the loop index. In these cases, the function can usually be implemented in a modestly sized lookup table that can often be placed in embedded memories. Functions that take arbitrary inputs can also be used in hardware lookup tables with interpolation. If you need more precision, you can try iterative or convergent techniques at the cost of more cycles. In these cases, the software compilation tools should make good use of existing FPGA resources.

What Skills Are Required to Use FPGA Acceleration?

Fortunately, hardware design from ESL tools has come a long way in the last few years. It is now possible to use C-based hardware languages to design the hardware portions of the system. You can easily generate FPGA hardware from the original software algorithms.

Design tools from Celoxica can compile or synthesize C descriptions directly into FPGA devices by generating an EDIF netlist or RTL description. These tools commonly provide software APIs to abstract away the details of the hardware/CPU connection from the application development. This allows you to truly treat the FPGA as a co-

processor in the system and opens doors to the use of FPGAs in many new areas such as high-performance computing, financial analysis, and life sciences.

The languages used to compile software descriptions to the FPGA are specifically designed to both simplify the system design process with a CPU and add the necessary elements to generate quality hardware (see Figure 3). Languages such as Handel-C and SystemC provide a common code base for developing both the hardware and software portions of the system. Handel-C is based on ANSI-C, while SystemC is a class library of C++.

Any process that compiles custom hardware from software descriptions has to deal with the following issues in either the language or the tool: concurrency, data types, timing, communication, and resource usage. Software is written sequentially, but efficient hardware must translate that code into parallel constructs (using potentially multiple hardware clocks) and implement that code using all of the proper hardware resources.

Both Handel-C and SystemC add simple constructs in the language to allow expert users control over this process, while maintaining a high level of abstraction for representing algorithmic designs. Software developers who understand the concepts of parallelism inherent to hardware will find these languages very familiar and can begin designing accelerated systems using the corresponding tools in the matter of weeks.

How Much Acceleration Can I Achieve?

Table 1 gives a few examples of the acceleration achieved in Celoxica customer projects. In another example, a CT (computed tomography) reconstruction-by-filtered-back-projection application takes in sets of 512 samples (a tomographic projection) and filters that data first through an FFT. It then applies the filter function to the frequency sample and applies an inverse FFT. These filtered samples are used to compute their contribution to each of the pixels in the 512 x 512 reconstructed image (this is the back-projection process). The process is repeated with the pixel values accumulated from each of the 360 projections. The fully reconstructed image is then displayed on a computer screen and stored to disk. Running on a Pentium 4 3.0 GHz computer, the reconstruction takes about 3.6 seconds. For the purposes of this project, the desired accelerated processing time target for reconstruction is 100 ms.

Profiling the application showed that 93% of the CPU run time is spent in the function that does the back projection, making that function the main target for acceleration. Analyzing the back-projection code showed the bottleneck clearly. For every pixel location in the final image, two filtered samples are multiplied by coefficients, summed, and accumulated in the pixel value. This function is invoked 360 times, and so the inner loop executes around 73 million times, each time requir-

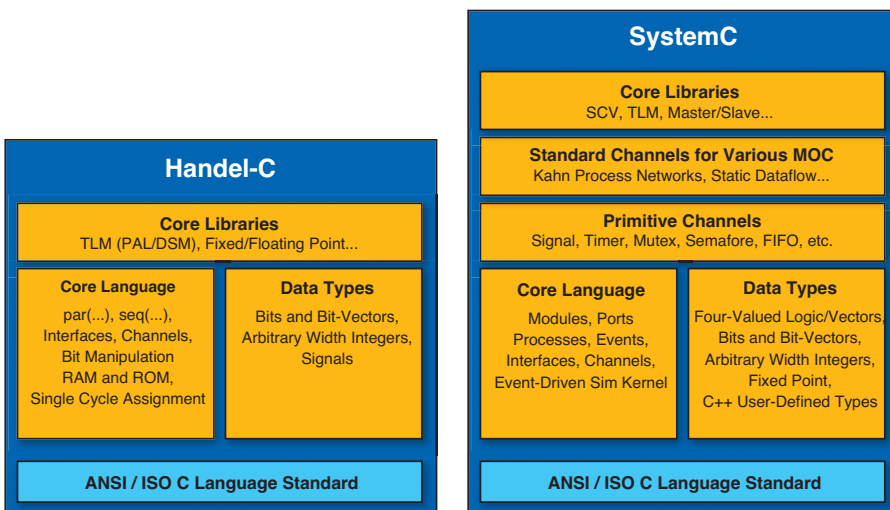


Figure 3 – Adding hardware capabilities

ing 3 reads and 1 write to memory, 3 multiplies, and several additions.

Fortunately, the back-projection algorithm lends itself well to parallel processing. Although the code was written using floating point, the integer nature of the input and output data made a fixed-point implementation perfectly reasonable. Working together, engineers from Celoxica and their customer accelerated the application, running on a standard PC, using a PCI add-in card with a Xilinx Virtex™-II FPGA and six banks of 512,000 x 32 SRAM. A sensible first-order partitioning of the application put the back-projection execution in the PLD, including the accumulation of the reconstructed image pixels, leaving the filtering and all other tasks running on the host processor.

The engineers designed a single instance of the fixed-point back-projection algorithm and validated it using the DK Design Suite. An existing API simplified the data transfer directly from the software application to the Xilinx FPGA over the PCI bus. They exploited fine-grained parallelism and rearranged some of the arithmetic to maximize the performance of the algorithm in hardware. The core unit was capable of pro-

cessing a single projection in about 3 ms running at a 66 MHz clock rate.

The fact that each projection could be computed independently meant that multiple back-projection cores could be running concurrently. The engineers determined that 18 cores would be enough parallelism to reach the performance goal comfortably. The software would pass 18 sets of 512 samples to the FPGA, which would store them in 18 individual dual-port RAMs. Both ports of these RAMs were interfaced to each back-projection core, allowing them to read the two samples required in a single cycle.


The cores were synchronized and the 18 contributions were fed into a pipelined adder tree. The final summed value was accumulated with the previous pixel value stored in one of the external SRAM banks. The accumulated value is written into the other SRAM bank and the two are read in ping-pong fashion on subsequent passes. With this architecture sample sets are transferred from software to hardware and the reconstructed image is passed back once when complete. The software was modified so that while one set of data was being processed in the FPGA, the next set was

being filtered by the host processor, further minimizing design time.

In the end, leveraging the features of FPGAs and modern ESL tools and taking advantage of the flexibilities provided by a software-based design flow, the example design was able to realize a 45x acceleration for a real-world application.

Conclusion

Accelerating complex software algorithms in programmable logic and integrating them into a system that contains processors or DSPs has never been easier or more affordable. Advanced FPGA architectures, high-speed interconnect such as PCI Express and HyperTransport, and the maturity of C-based ESL design tools reduce the design burden on developers and open up new possibilities for the application of FPGA technology. Balancing the needs of power, performance, design time, and cost is driving the FPGA into new market applications.

For more information about ESL design solutions for Xilinx FPGAs, low-cost starter kits, and FPGA-based acceleration benchmarks, visit www.celoxica.com/xilinx. 

Application	Hardware Co-Processor	Software Only
Hough and Inverse Hough Processing	2 sec of Processing Time @ 20 MHz 370x Faster	12 Minutes Processing Time Pentium 4 - 3 GHz
AES 1 MB Data Processing/Cryptography Rate Encryption Decryption	424 ms / 19.7 MBps 424 ms / 19.7 MBps 13x Faster	5,558 ms / 1.51 MBps 5,562 ms / 1.51 MBps
Smith-Waterman ssearch34 from FASTA	100 sec FPGA processing 64x Faster	6461 sec Processing Time Opteron
Multi-Dimensional Hypercube Search	1.06 sec FPGA @ 140 MHz Virtex-II Device 113x Faster	119.5 sec Opteron - 2.2 GHz
Callable Monte-Carlo Analysis 64,000 Paths	10 sec of Processing @ 200 MHz FPGA System 10x Faster	100 sec Processing Time Opteron - 2.4 GHz
BJM Financial Analysis 5 Million Paths	242 sec of Processing @ 61 MHz FPGA System 26x Faster	6300 sec Processing Time Pentium 4 - 1.5 GHz
Mersenne Twister Random Number Generation	319M 32-bit Integers/sec 3x Faster (Bus Bandwidth BW Limited - Processing ~ 10-20x Ratio)	101M 32-bit Integers/sec Opteron - 2.2 GHz

Table 1 – Augmenting processor performance